设计模式III

（OOP）

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/hml

课程团队：刘知远 姚海龙 黄民烈

上期要点回顾

- 语言集成查询LINQ解析器
- 迭代器模式
- 模板模式
- 策略模式

本讲内容提要

- 单例（Singleton）模式
- 适配器（Adapter）模式
- 装饰器（Decorator）模式

单例模式
Singleton

例子：全局计数器

- 一个在多个地方被调用的计数器
  - 调用addCount则计数器+1
  - 调用getCount则输出计数器数值

```
void func() {
    addCount();
}

int main() {
    addCount();
    func();
    cout << getCount() << endl;     // 结果为2
}
```

- 如何实现addCount和getCount函数？

全局变量

- 一种最直接的方法：

```
int count = 0;

void addCount() {
    count += 1;
}

int getCount() {
    return count;
}
```

- 用户可能访问到count并修改数据，不安全
- 好的设计应当避免全局变量

静态方法

- 定义一个类，将函数实现为静态（static）方法：

```
class Counter {
private:
    //定义为私有的静态成员，使得用户无法直接访问
    static int count = 0;
public:
    static void addCount() {
        count += 1;
    }
    static int getCount() {
        return count;
    }
};
```

```
Counter::addCount();
cout << Counter::getCount() << endl;
```

静态方法 & 虚函数

- 如果我们有多种不同的Counter……
  - ```
    class BaseCounter {
    public:
        static virtual void addCount() = 0;
        // ...
    };
    ```
  - ```
    class SimpleCounter : public BaseCounter {
        static int count = 0;
    public:
        static void addCount() {
            count += 1;
        }
        // ...
    };
    ```
  - ```
    class NotSimpleCounter : public BaseCounter ...
    ```
- 可以这么写吗?
    ```

静态方法 + 虚函数 = 编译错误

- 静态方法不可以是虚的!
- 静态方法只与类相关而不与实例相关，调用时必须知道类名

- 能否不使用静态方法达到我们的目的?
  - 我们想用静态方法实现的是：无论在何处调用，都会访问到相同的函数和变量
  - 也即，类存在全局唯一的实例

单例模式

- 所谓单例，就是只能构造一份实例的类

```
class Counter {
    // 显式删除拷贝构造函数与赋值操作符
    Counter(const Counter &) = delete;
    void operator =(const Counter &) = delete;

    int count;
    Counter() { count = 0; } // 将构造函数设为私有
    static Counter _instance; // 全局唯一的实例

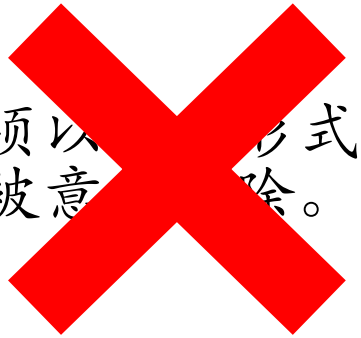
public:
    static Counter &instance() {
        return _instance;
    }
    // 成员函数而非静态方法
    void addCount() { count += 1; }
    int getCount() { return count; }
};
```

单例模式

- 调用单例
  - ```
    class Counter { ... };
    // 定义类中的静态成员，单例在此被初始化
    Counter Counter::_instance;
    ```
  - ```
    int main() {
        // 由于删去了拷贝构造函数，必须存为引用（指针也可以）
        Counter &c = Counter::instance();
        c.addCount();
        cout << c.getCount() << endl;
    }
    ```
- 单例模式封装了全局性的变量，无需多次实例化，很好的满足了我们的需要
- 缺点：类静态成员和全局变量一样，初始化顺序不确定

惰性初始化（Lazy Initialization）

- 能否在使用时再构造单例实例？
  - ```
    class Counter {
        // ...
    public:
        static Counter &instance() {
            static Counter _instance;
            return _instance;
        }
        // ...
    };
    ```
- 在第一次调用instance方法时才会构造单例
    ```

单例模式：陷阱！

- 刚才的实现真的是没问题的吗？
- 需要避免的情况：
  - 实例被重复构造
    - 由于构造函数为private，且拷贝构造函数、赋值操作符被显式删除，故无法重复构造。
  - 实例被意外删除
    - 由于返回值必须以引用形式存储（而非指针或实例），故无法被意外删除。

单例模式：陷阱！

- 刚才的实现真的是没问题的吗？
- 需要避免的情况：
  - 实例被重复构造
    - 由于构造函数为private，且拷贝构造函数、赋值操作符被显式删除，故无法重复构造。
  - 实例被意外删除
    - 由于返回值必须以[illegible]形式存储（而非指针或实例），故无法被意[illegible]除。



错！

单例模式：陷阱！

- 考虑下面的代码：
  - ```
    Counter &c = Counter::instance();
    delete &c;     // 可以成功执行!
    c.addCount(); // 运行时错误
    ```
- 应当把析构函数也设为**private**!
  - ```
    class Counter {
    private:
        ~Counter() {}
    // ...
    }
    ```
  - ```
    error: calling a private destructor of class 'Counter'
        delete &c;
                ^
    ```
    ```

单例模式 + 虚函数

- 回到一开始的问题：
  - ```
    class BaseCounter {
    public:
        virtual void addCount() = 0;
        virtual int getCount() = 0;
    };
    ```
  - ```
    class SimpleCounter : public BaseCounter {
        // ...单例相关的一大堆逻辑...(显式删除复制构造函数等)
        int count;
        SimpleCounter() { count = 0; }
    public:
        virtual void addCount() { count += 1; }
        virtual int getCount() { return count; }
    };
    ```
  - ```
    class NotSimpleCounter : public BaseCounter ...
    ```
    ```

单例模式 + 虚函数

```
void doStuff(BaseCounter *counter) {
    counter->addCount();
    counter->addCount();
    cout << counter->getCount() << endl;
}
int main() {
    doStuff(&SimpleCounter::instance());      //  2
    doStuff(&NotSimpleCounter::instance());   // ...
    doStuff(&SimpleCounter::instance());      //  4
}
```

- 唯一的不便：每个派生类都要重复实现单例相关的逻辑
- 为什么？因为基类不知道派生类的类别，无法在基类中声明派生类的静态变量
- 能否让基类知道这一信息呢？

“奇特的递归模板模式”

- “奇特的递归模板模式”: Curiously Recurring Template Pattern (CRTP)
  -
```
template <class Derived>        // 模板参数为派生类类型
class Singleton {
    Singleton(const Singleton &) = delete;
    void operator =(const Singleton &) = delete;
protected:
    Singleton() {}
    virtual ~Singleton() {}
public:
    static Derived &instance() {  // 魔法在此发生
        static Derived _instance;
        return _instance;
    }
};
```

CRTP + 多重继承

- 基于Singleton类实现计数器派生类:
  - class SimpleCounter : public BaseCounter,
    public Singleton<SimpleCounter> {

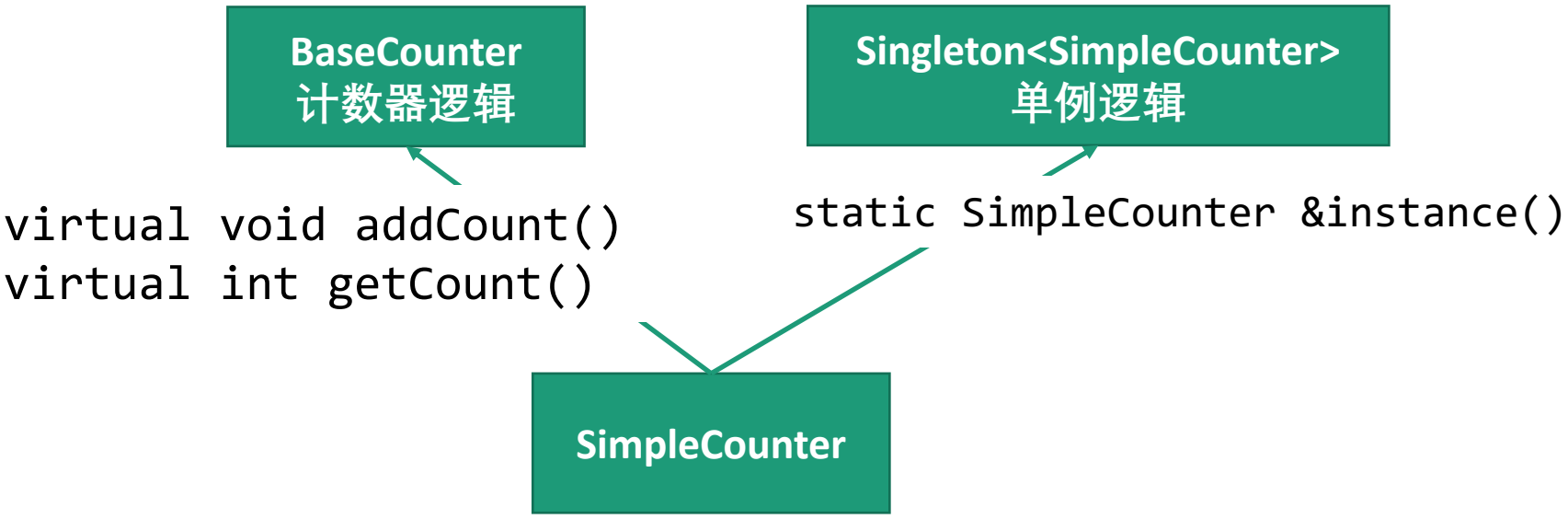
```
// 友元声明是必要的，因为Singleton类需要访问派生类的
// 构造函数，而为了实现单例，构造函数是私有的
friend class Singleton<SimpleCounter>;
// ... 只需实现计数器逻辑即可
```

    }
  - 友元：Singleton<SimpleCounter>可以访问SimpleCounter的私有成员

```
static Derived &instance() {  //Singleton<SimpleCounter>
    static Derived _instance;
    return _instance;
}
```

- 任何类只需继承Singleton<Derived>，即可自动生成单例接口

CRTP + 多重继承



- 注意，不能直接将Singleton类的逻辑实现在BaseCounter类中，否则BaseCounter将成为一个模板类，不存在一个通用的“基类指针”

关于CRTP

- CRTP是实现多态的另一种方式
- 与虚函数不同，本质上实现的还是编译期多态
- 考虑一个接受任意派生类实例的函数，在函数中调用实例的方法：
  - 使用虚函数实现：运行时通过虚函数表寻找调用的方法
  - 使用CRTP实现：函数需要被实现为模板函数，编译时由编译器为每种被调用的派生类进行模板实例化

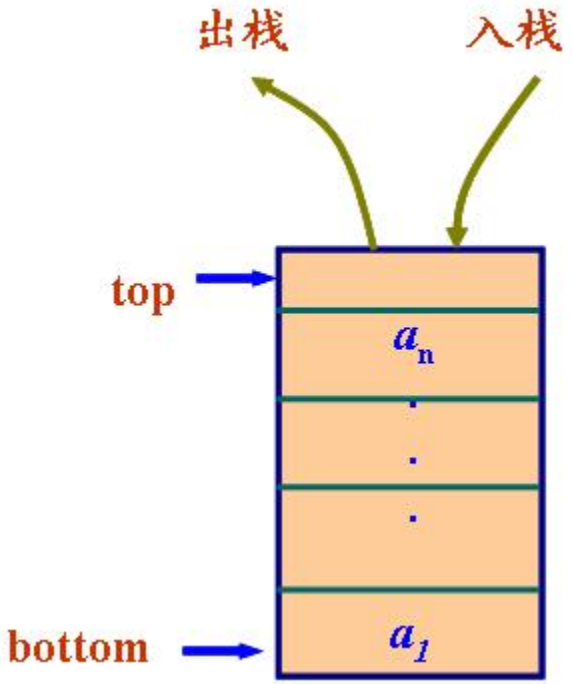
关于单例模式

- 单例模式是存在争议的一种设计模式
- 优点：
  - 实现似乎比较简单
  - 以相对安全的形式提供可供全局访问的数据
- 缺点：
  - 难以完全正确地实现
  - 违反单一职责原则
  - 过度使用这一方法会使得实际的依赖关系变得隐蔽

适配器
Adapter

一个简单例子—栈

- 功能类似数组
- 元素访问规则有所不同，是“后进先出”（Last-In-First-Out）
- 简单起见，只支持int类型的元素



简单实现

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

简单实现

```
class MyStack : public Stack{

private:
    int *m_data; const int m_size; int m_top;

public:
    //构造函数
    MyStack(size) : m_size(size), m_top(-1), m_data(NULL) {
        if (m_size > 0) m_data = new int[m_size];
    }
    //析构函数
    virtual ~MyStack() {
        if (m_data) delete [] m_data;
    }
    //满栈检测
    bool full() {
        return m_size <= 0 || (m_top+1) == m_size;
    }
```

简单实现

```
    //空栈检测
    bool empty() {
        return m_top < 0;
    }
    //入栈
    void push(int i) {
        if (m_top+1 < m_size) m_data[++ m_top] = i;
    }
    //出栈
    void pop() { if (!empty()) --m_top;  }
    //获取堆栈已用空间
    int size() { return m_top+1; }
    //获取栈头内容
    int top() {
        if (!empty())
            return m_data[m_top];
        else
            return INT_MIN;
    }
};
```

简单实现

```
int main(int argc, char *argv[]) {

    //创建一个最多放置10个元素的栈
    MyStack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

```
4
3
2
1
```

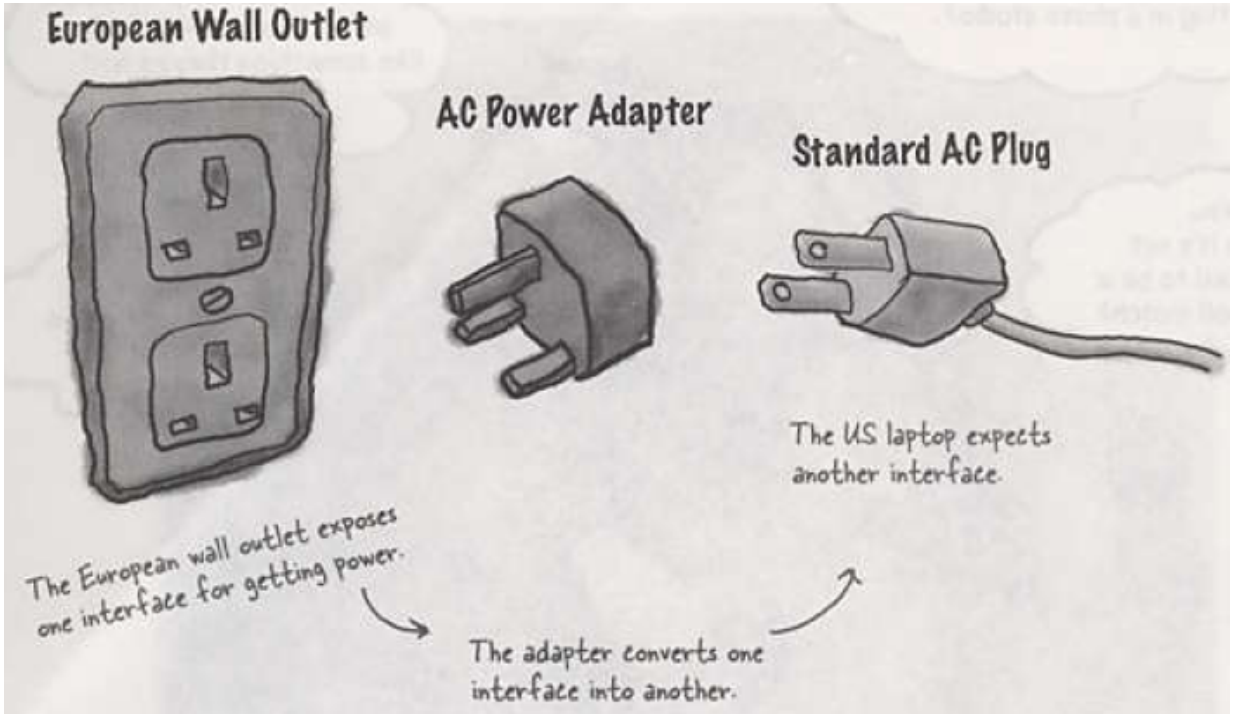
这样的代码有什么问题？

STL vector

- 工作量太大(OOP思想之一：复用)
- 需要我们自行管理内存(容易出错)

- 事实上STL中有vector这个容器
- vector提供了如下方法:
  - push_back()
  - size()
  - back()
  - pop_back()

分析



- Vector
  - 功能上满足要求（内存管理，元素插入弹出）
  - 但是接口不一致
- 需要进行接口的“转换”

适配器

- 考虑生活中一种常见的情况：
  - 有手机、手机充电线，要给手机充电。
  - 充电线只能插在USB接口上进行充电。
  - 但是现在只有220V的插座可以供电。
  - 所以需要用一个转接头将220V插座和USB口衔接。

- 这里的转接头实际上就是一种现实中的适配器

适配器

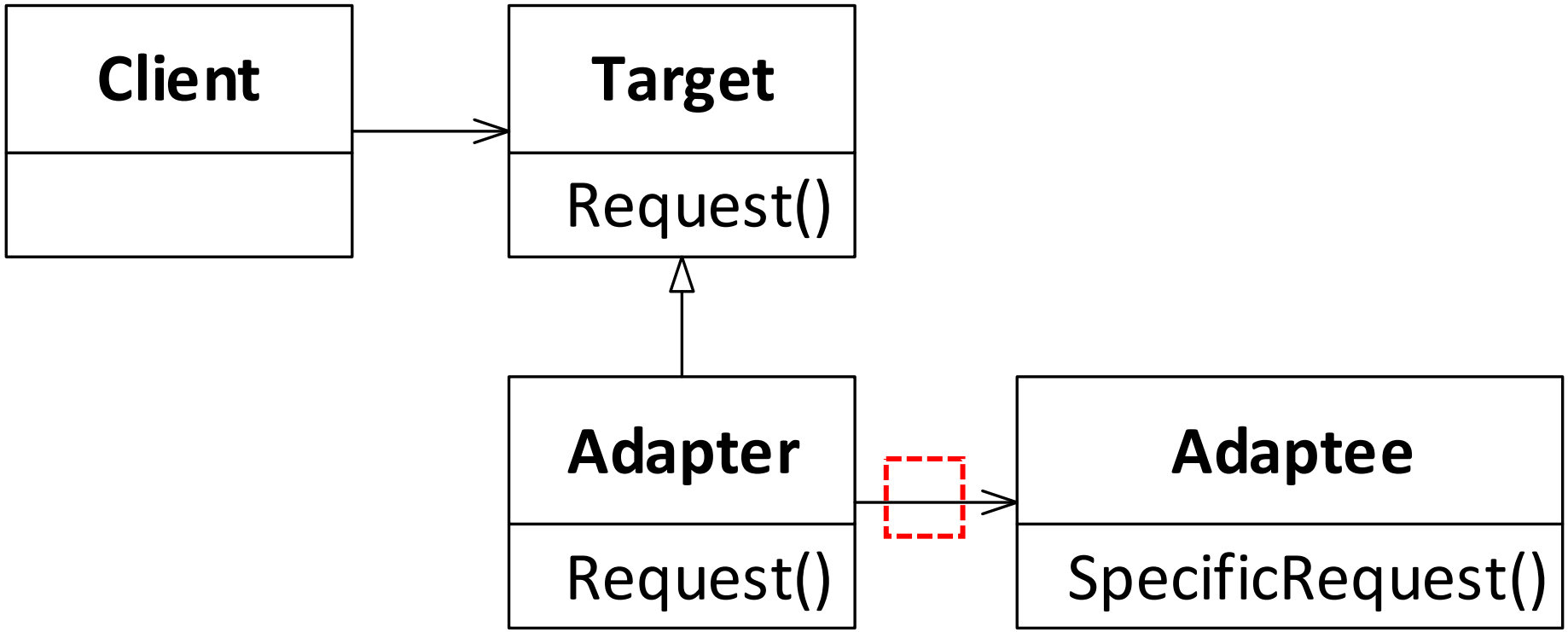
概述

- 适配器模式将一个类的接口转换成客户希望的另一个接口，从而使得原本由于接口不兼容而不能一起工作的类可以在统一的接口环境下工作。

结构

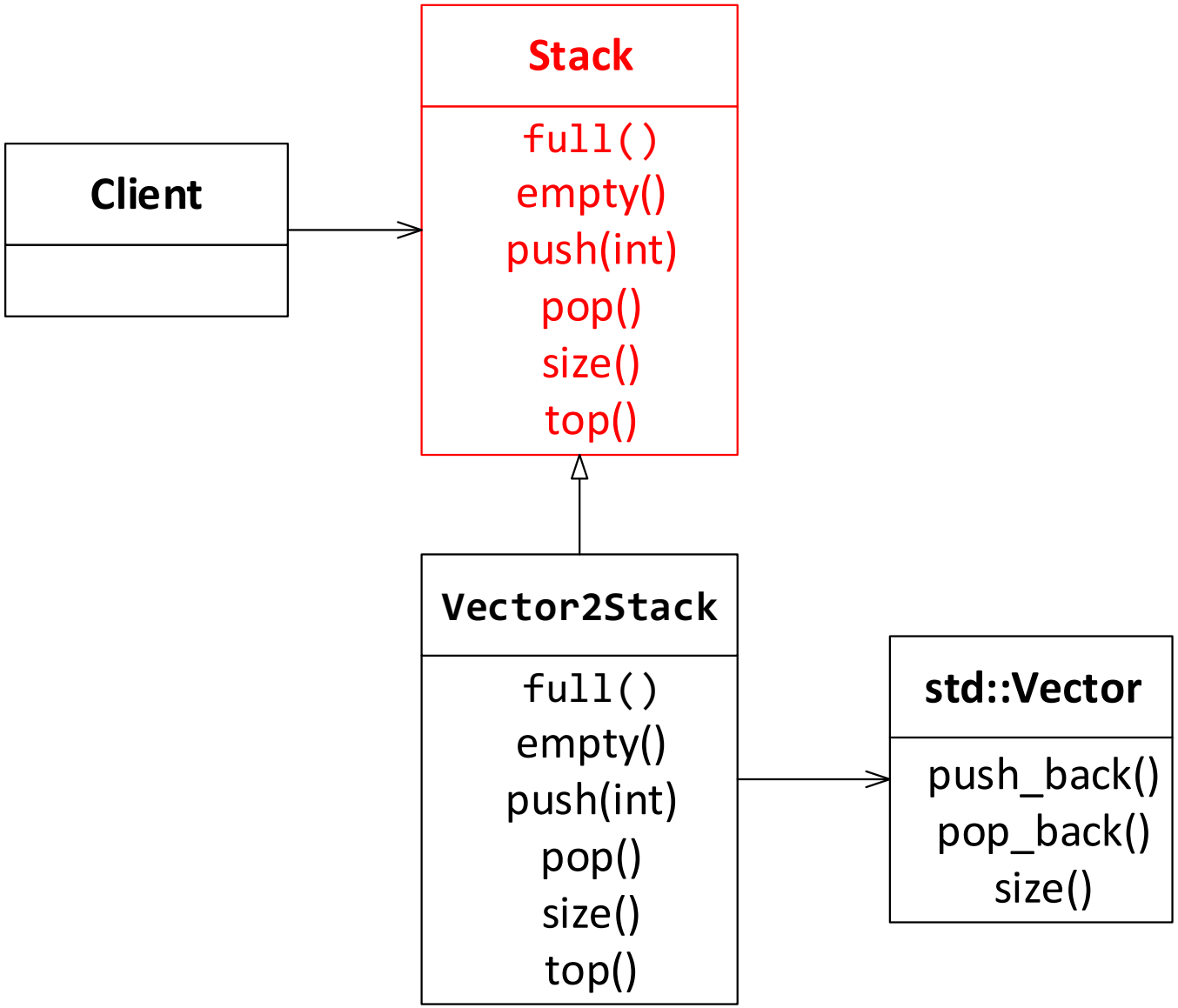
- 目标（Target）：客户所期待的接口。
- 需要适配的类（Adaptee）：需要适配的类。
- 适配器（Adapter）：通过包装一个需要适配的类，把原接口转换成目标接口。

适配器——实现一



使用组合实现适配，称作对象适配器模式

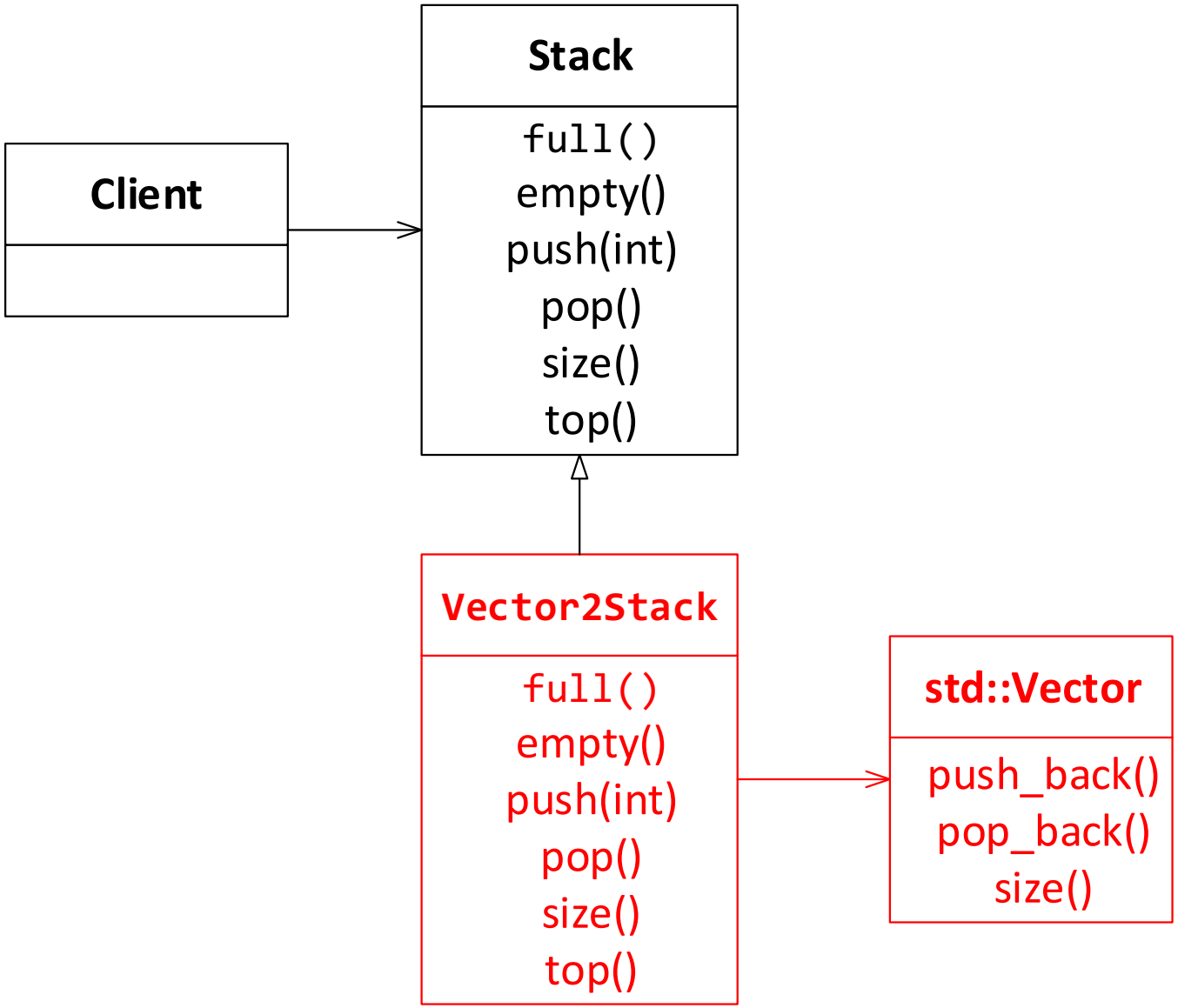
适配器基类定义


Stack
full()
empty()
push(int)
pop()
size()
top()
Client
Vector2Stack
full()
empty()
push(int)
pop()
size()
top()
std::Vector
push_back()
pop_back()
size()

适配器——实现一

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

组合方式实现适配器模式



适配器——实现一

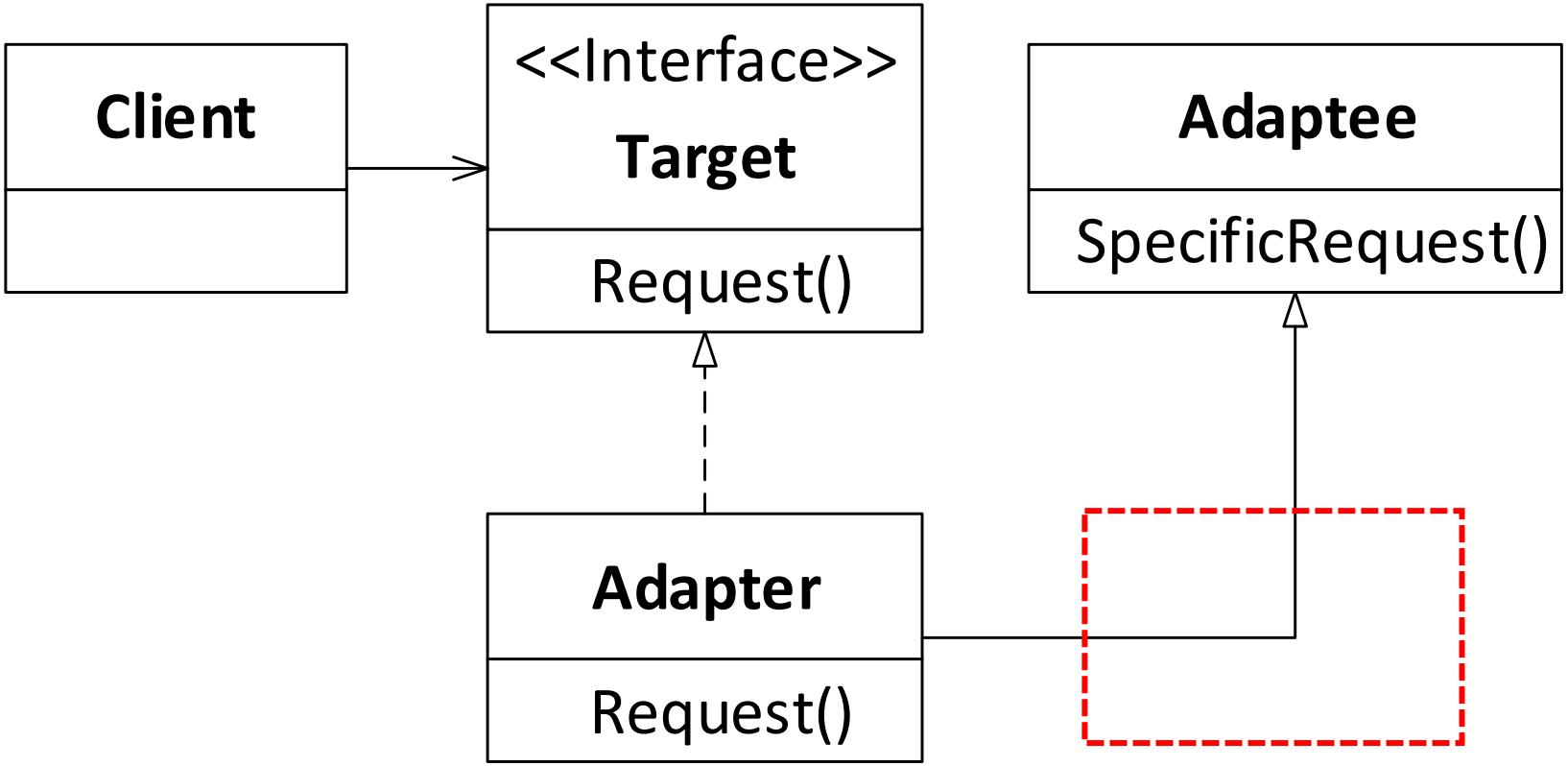
```
class Vector2Stack : public Stack{
private:
    std::vector<int> m_data; //将vector的接口组合进来实现具体功能
    const int m_size;
public:
    Vector2Stack(int size) : m_size(size) { }
    bool full() { return (int)m_data.size() >= m_size; }   //满栈检测
    bool empty() { return (int)m_data.size() == 0; }   //空栈检测
    void push(int i) { m_data.push_back(i); }       //入栈
    void pop() { if (!empty()) m_data.pop_back(); }//出栈
    int size() { return m_data.size(); }            //获取堆栈已用空间
    int top() {                                     //获取栈头内容
        if (!empty())
            return m_data[m_data.size()-1];
        else
            return INT_MIN;
    }
};
```

适配器——实现一

```
int main(int argc, char *argv[]) {

    Vector2Stack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        std:: cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

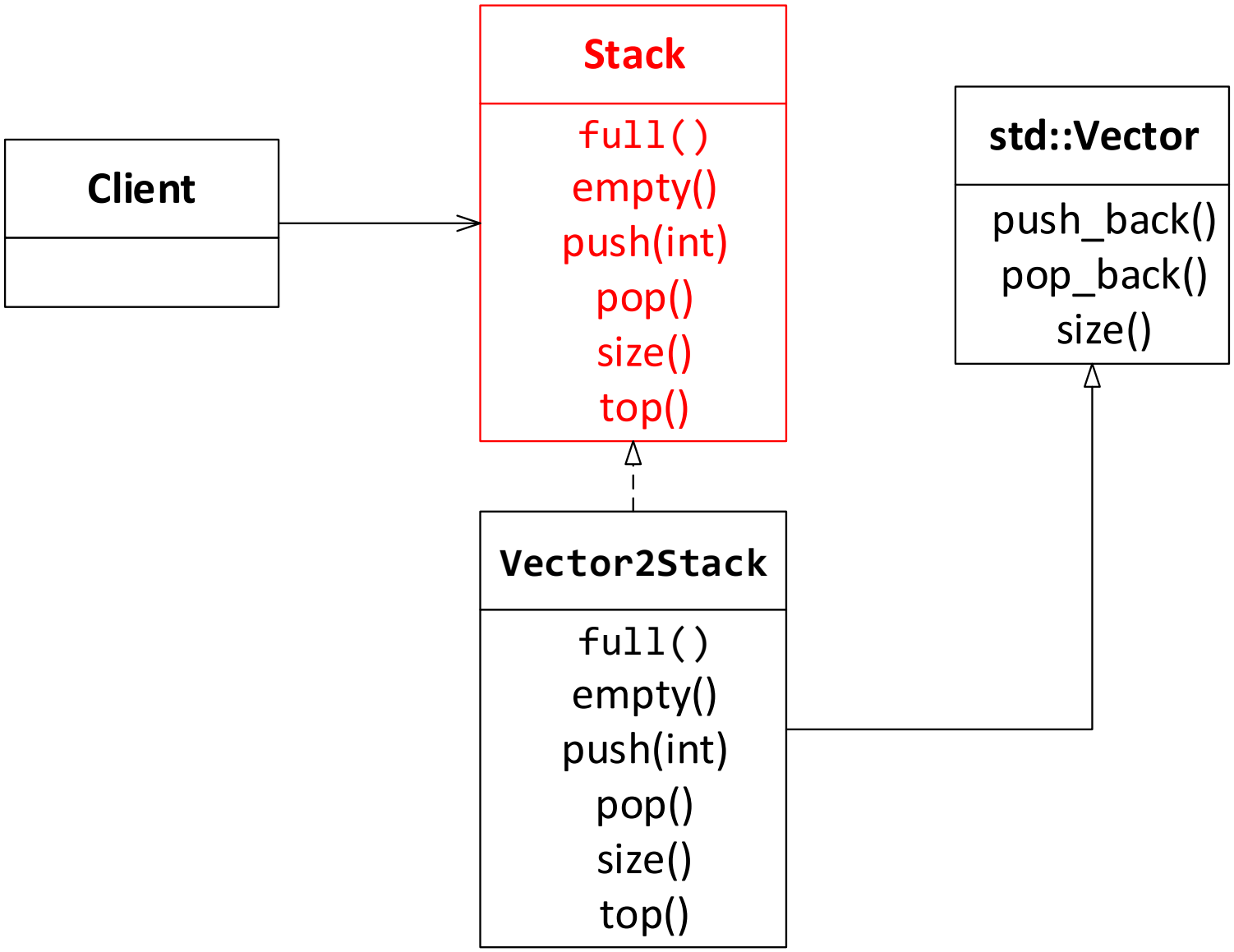
```
4
3
2
1
```

适配器——实现二



使用继承实现适配，称作类适配器模式

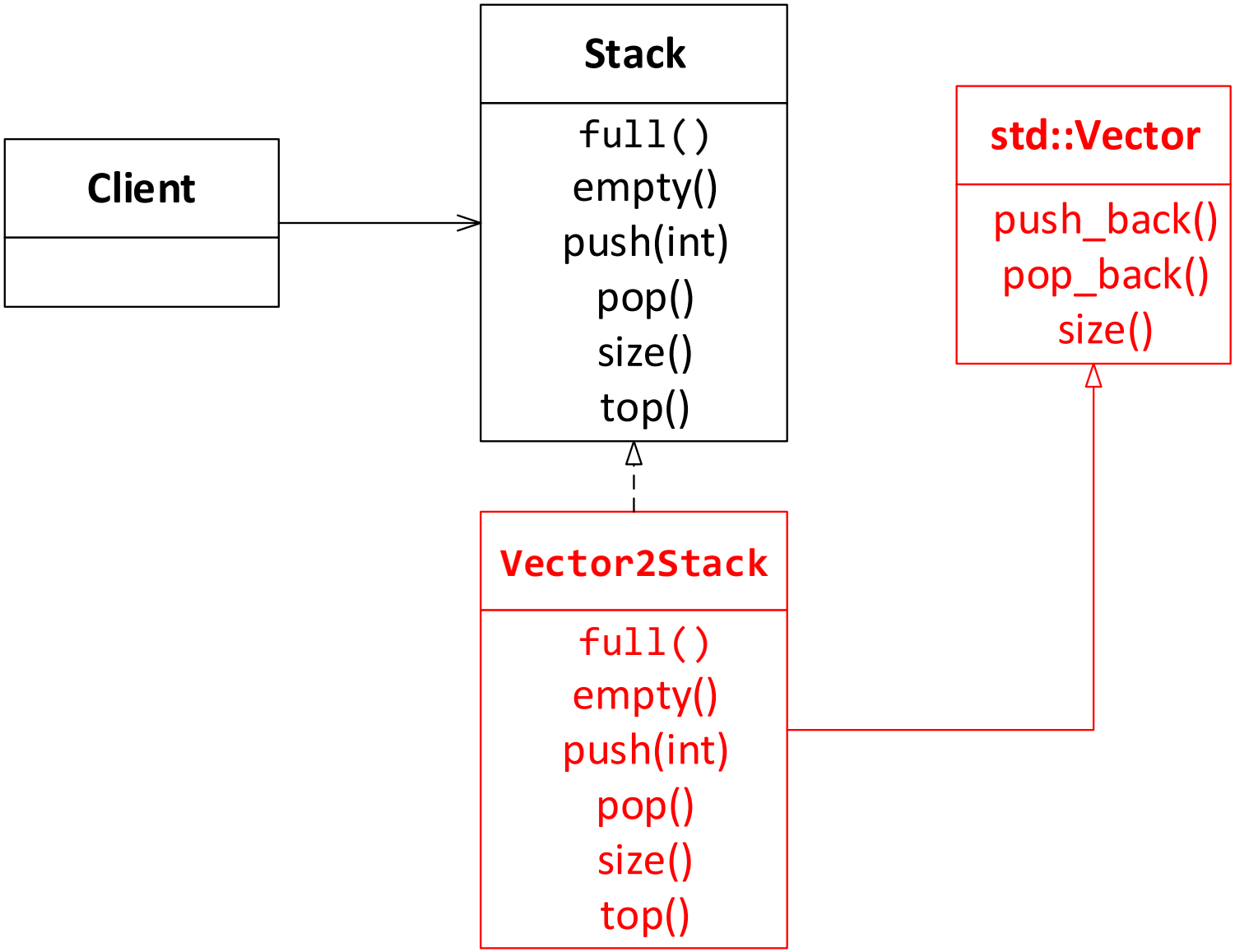
适配器接口定义


Client
Stack
full()
empty()
push(int)
pop()
size()
top()
std::Vector
push_back()
pop_back()
size()
Vector2Stack
full()
empty()
push(int)
pop()
size()
top()

适配器——实现二

```
#include <cstring>
#include <cstdio>
#include <vector>
#include <iostream>

//堆栈基类
class Stack {
public:
    virtual ~Stack() { }
    virtual   bool full() = 0;
    virtual   bool empty() = 0;
    virtual   void push(int i) = 0;
    virtual   void pop() = 0;
    virtual   int size() = 0;
    virtual   int top() = 0;
}
```

继承方式实现适配器模式



适配器——实现二

```
//直接继承vector并改造接口，采用私有继承可以使得外界只能接触到Vector2Stack中的接口

class Vector2Stack : private std::vector<int>, public Stack {
public:
    Vector2Stack(int size) : vector<int>(size) { }
    bool full() { return false; }
    bool empty() { return vector<int>::empty(); }
    void push(int i) { push_back(i); }
    void pop() { pop_back(); }
    int size() { return vector<int>::size(); }
    int top() { return back(); }
};
```

适配器——实现二

```
int main(int argc, char *argv[]) {

    Vector2Stack stack(10);

    //压入1,2,3,4
    for (int i = 1; i < 5; i++)
        stack.push(i);

    //逐个弹出
    for (int i = 0; i < 4; i++) {
        std::cout << stack.top() << "\n";
        stack.pop();
    }

    return 0;
}
```

```
4
3
2
1
```

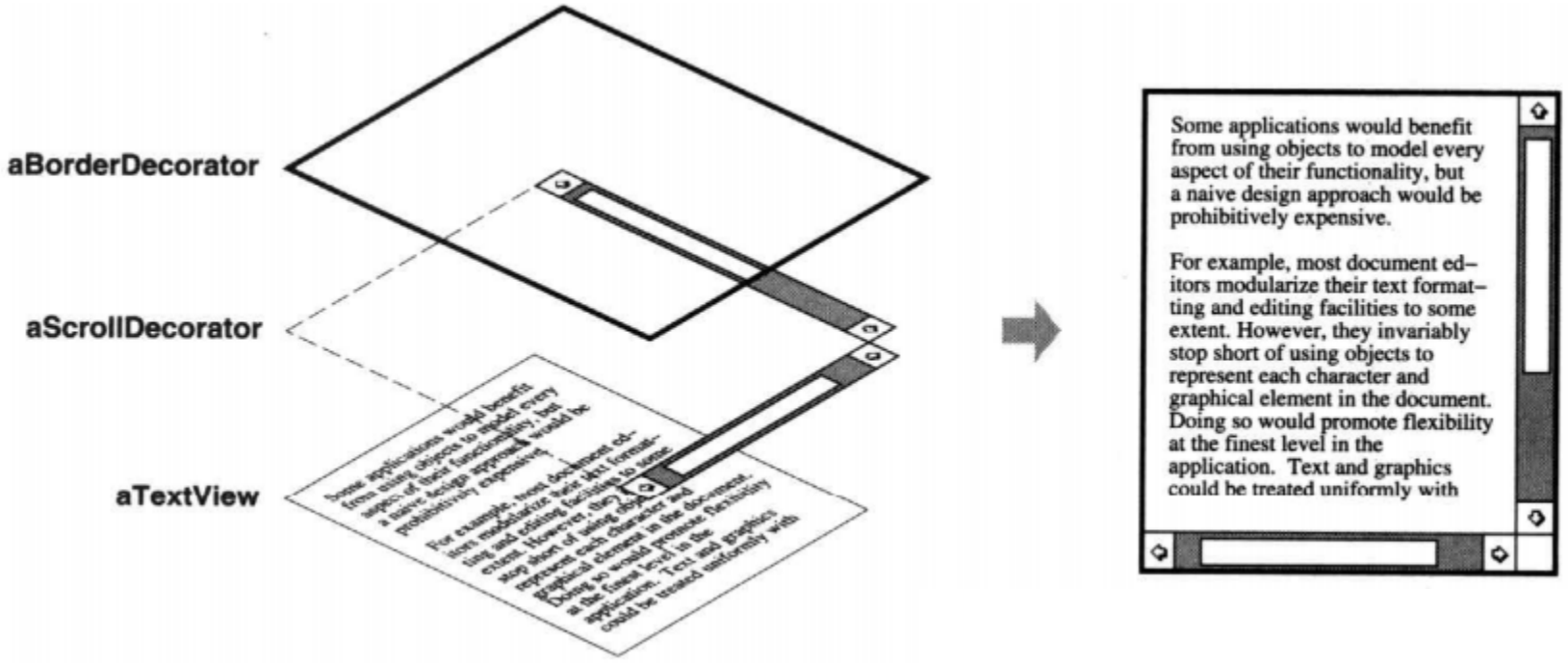
适配器

- 优点
  - 通过适配器，客户端可以用统一接口调用各种复杂的底层工作类
  - 复用了现有的类，提高代码复用率
  - 将目标类和适配者类解耦，通过引入一个适配器类包装现有的适配者类以满足新接口需求，无需修改原有代码
- 适用场景举例
  - 系统需要复用已有的类，但这些类的接口不符合系统的接口
  - 接入第三方组件，但组件接口定义与自身定义不同
  - 旧系统开发的类已经实现了一些功能，但是客户端只能以新接口的形式访问，且我们不希望手动更改原有类

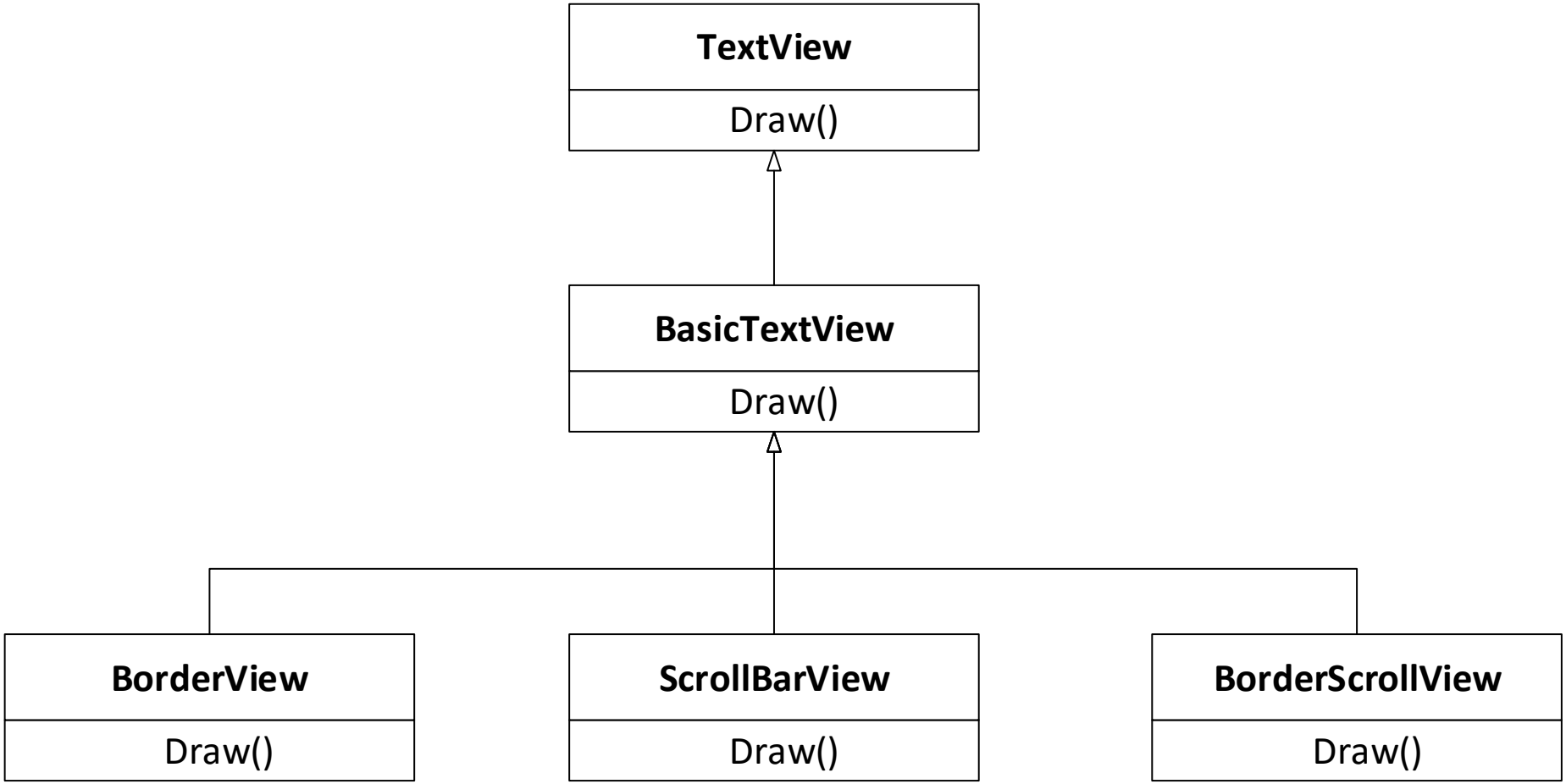
装饰器
Decorator

例子

- 有一个对象TextView，在窗口中显示文本
- 希望接口不变，增加滚动条、边框、……



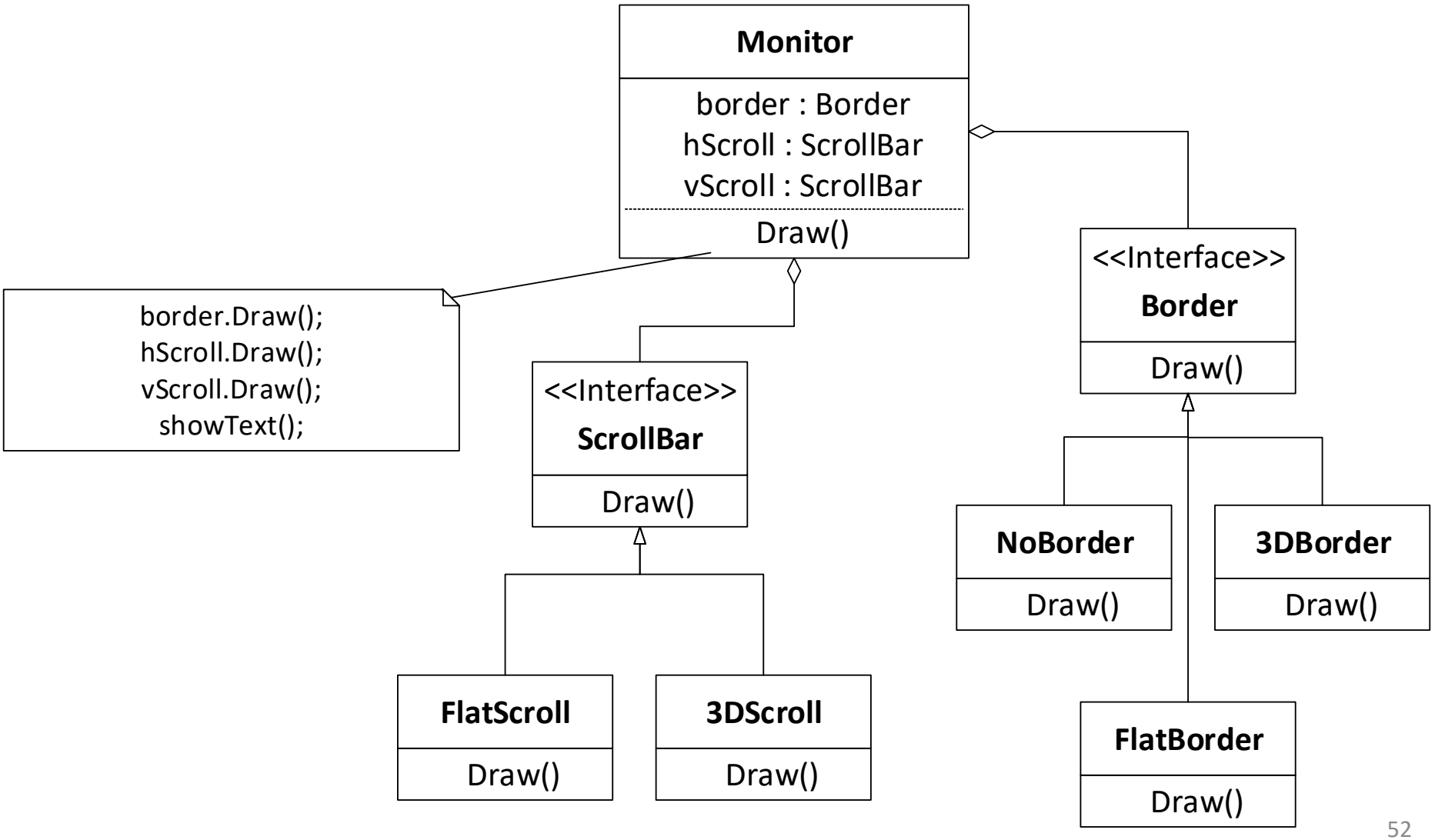
继承


TextView
Draw()
BasicTextView
Draw()
BorderView
Draw()
ScrollBarView
Draw()
BorderScrollView
Draw()

继承

- 使用继承
- 依靠多态实现功能的变化
- 问题
  - 随着功能的变多，继承类的数量急剧膨胀，其最大派生类的数目可以是所有功能的组合数
  - 如果TextView的基类增加新的接口，那么所有的派生类都需要进行修改

策略

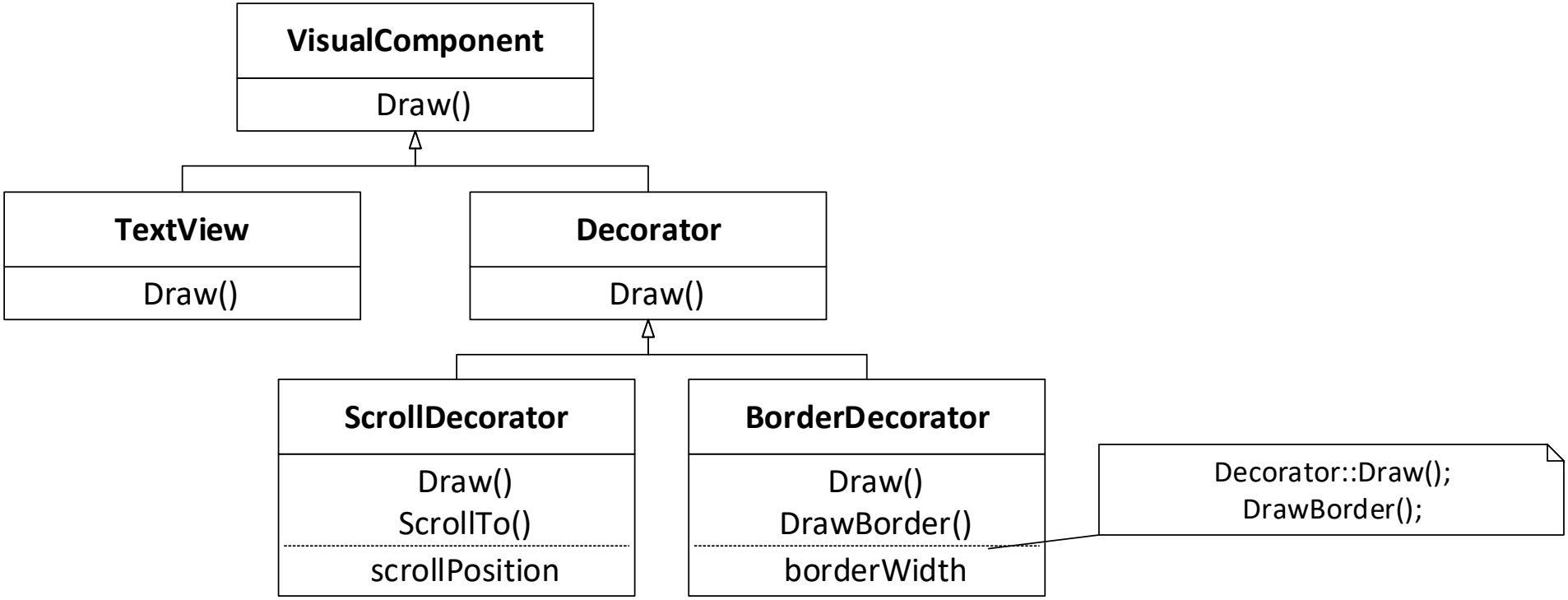

Monitor
border : Border
hScroll : ScrollBar
vScroll : ScrollBar
Draw()
border.Draw();
hScroll.Draw();
vScroll.Draw();
showText();
<<Interface>>
ScrollBar
Draw()
FlatScroll
Draw()
3DScroll
Draw()
<<Interface>>
Border
Draw()
NoBorder
Draw()
3DBorder
Draw()
FlatBorder
Draw()

策略

- 用组合替代继承
- 问题
  - 策略的个数是基类中预先定义好的，如基类中定义了边框和滑动条，那么策略模式只能实现不同的边框与滑动条功能的组合。
  - 如果我要再增加一个滚动条和边框之外的新功能，那么就要修改基类，在基类中增加策略个数和新的方法。这样对整体框架的改动是我们不乐意见到的。

装饰器

- 创建了一个装饰类，用来包装原有的类，并在保持类方法完整性的前提下，提供了额外的功能。
- 且装饰类与被包装的类继承于同一基类，这样装饰之后的类可以被再次包装并赋予更多功能。



代码

```
#include <iostream>

using namespace std;

//所有View的基类
class Component {
public:
    virtual ~Component() { }
    virtual void draw() = 0;
};

//一个基本的TextView类
class TextView : public Component {
public:
    void draw() {
        cout << "TextView." << endl;
    }
};
```

代码

```
//装饰器的核心内涵在于用装饰器类整体包裹改动之前的类，以保留原来的全部接口
//在原来接口保留的基础上进行新功能扩充

class Decorator : public Component {
    //这里一个基类指针可以让Decorator能够以递归的形式不断增加新功能
    Component* _component;
public:
    Decorator(Component* component) : _component(component) {
    }
    virtual void addon() = 0;
    void draw() {
        addon();
        _component -> draw();
    }
};
```

代码

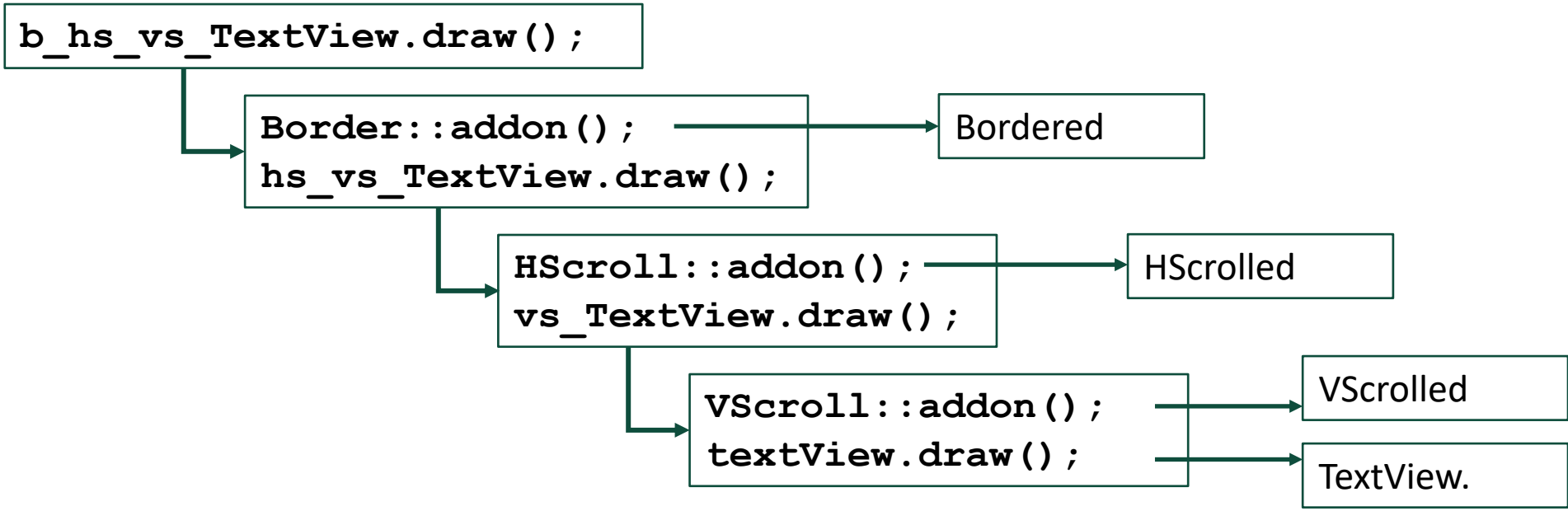
```
//包裹原Component并扩充边框
class Border : public Decorator {
public:
    Border(Component* component) : Decorator(component) { }
    void addon() { cout << "Bordered "; }
};

//包裹原Component并扩充水平滚动条
class HScroll : public Decorator {
public:
    HScroll(Component* component): Decorator(component) { }
    void addon() { cout << "HScrolled "; }
};

//包裹原Component并扩充垂直滚动条
class VScroll : public Decorator {
public:
    VScroll(Component* component): Decorator(component) { }
    void addon() { cout << "VScrolled "; }
};
```

代码

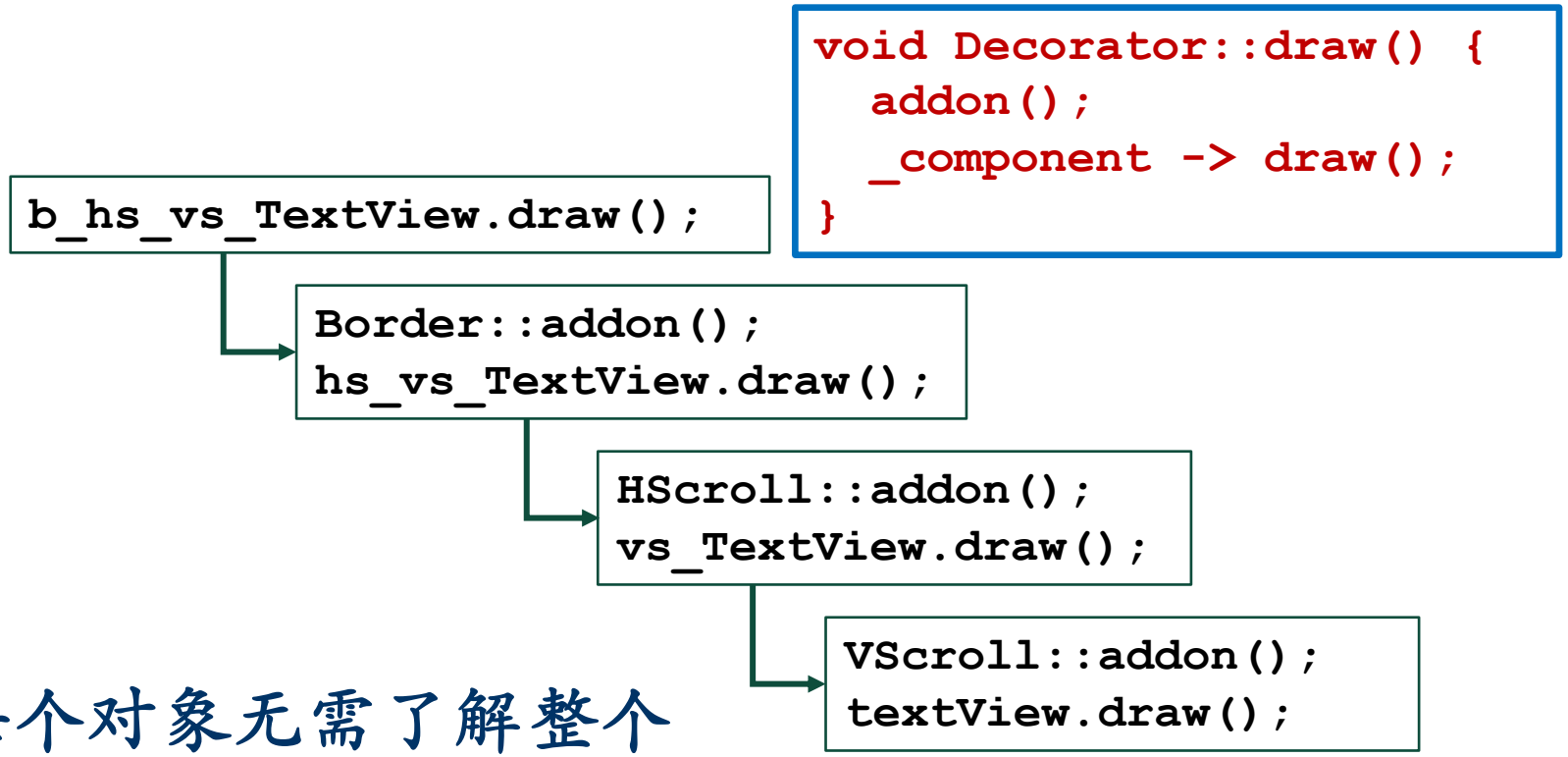
```
int main(int argc, char** argv) {
    //基础的textView
    TextView textView;
    //在基础textView上增加滚动条
    VScroll vs_TextView(&textView);
    //在增加垂直滚动条的基础上增加滚动横条
    HScroll hs_vs_TextView(&vs_TextView);
    //在增加水平与垂直滚动条之后增加边框
    Border b_hs_vs_TextView(&hs_vs_TextView);
    b_hs_vs_TextView.draw();
    return 0;
}
```

运行过程与结果

```
Bordered HScrolled VScrolled TextView.
```



调用的链式关系



- 每个对象无需了解整个链的全貌
- 每一次都是将之前的版本完全包裹住，再增加新的功能。换句话说，有多少个新功能就包裹几次

装饰与策略

- 相同点
  - 通过对象的组合修改对象的功能
  - 以组合替代简单继承，更加灵活，减少冗余
- 不同点

策略	装饰
• 修改对象功能的内核（行为） • 组件必须了解有哪些需要选择的策略，侧重于功能选择	• 修改对象功能的外壳（结构） • 组件无需了解有哪些可以装饰的内容，侧重于功能组装

装饰与适配器

- 都可以改变对象的行为

装饰	适配器
• 为被装饰对象增加额外的行为 • 不影响被装饰对象的原有功能 • 经常多重嵌套装饰	• 适配器可能会改变接口 • 适配器可能会改变功能 • 少见多重嵌套

本节课

- 介绍了单例、适配器、装饰模式。
  - 单例模式负责管理全局访问的对象
  - 适配器模式在类与类之间进行转接，能够了类的复用度与灵活性
  - 装饰器模式可以动态扩展被装饰类的功能，并留有接口进行持续扩展
- 核心就在于明确控制权限，尽可能的复用代码，以最小的代价支持新功能的增加。
- 拓展阅读
  - 《设计模式》3.5(单件)，4.1(适配器)，4.4(装饰)

感谢同学们的坚持和努力!

给我们每一个自己鼓掌!

儿童节快乐！